Analysis of Linux® Test Project's Kernel Code Coverage

Manoj Iyer
IBM, Austin TX, 78759
manjo@austin.ibm.com

Abstract

The Linux Test Project (LTP) is a large collection of automated and semi-automated tests for testing the Linux kernel. The coverage analysis effort aims at identifying the areas in the kernel that are currently tested and also identify the areas that require attention.

This article has several goals. First, describe the methods used to collect kernel code coverage data, convert this data to a convenient format that is suitable for visualization and analyze kernel code coverage provided by the test-suite. Second, present our preliminary results and discuss their implications. Finally, discuss the scope for future work.

The data required for this analysis was collected using the open-source profiling tool named gcov. The raw data was processed using tools written in gawk, C and PERL to generate HTML pages. Certain baseline assumptions were made in order to qualify coverage for a certain subsystem of the kernel. This HTML document is made available at the Linux Test Project's web site http://ltp.sf.net/.

1. Introduction

The analysis of the kernel test-suites Linux kernel code coverage is an effort to identify the areas in the kernel code that are impacted by the execution of the tests in the test-suite. The result of this study is published to the Linux Open Source community with an intention to demonstrate the effectiveness of this test-suite and also to invite community participation in our effort to make Linux better.

This article describes the various models, methodologies, tools, and data formats that were used to facilitate our analysis. In our analysis we do not attempt to make any claims with regards to the effectiveness of the test-suite, but we present statistics that can help kernel developers to use this test-suite and also contribute test-cases to areas that require focus.

The kernel test-suite maintained by LTP is widely used by the Linux kernel developer community, distributors of Linux operating systems, Linux kernel hackers and enthusiasts, for unit testing, integration testing, system testing, regression testing etc. The LTP uses this test-suite to test new releases of the Linux kernel and maintains an archive of results on its project website.

There has been increasing growth in the use of this test-suite. The community has shown keen interest in improving this test-suite, making it a robust and a standard validation tool for the Linux kernel. Also, the maintainers of the Linux kernel have shown interest in the test results published by LTP on release candidate and pre-release kernels. Our motivation for analyzing kernel coverage by LTP kernel test-suite is then straightforward and obvious, to gain understanding of the current coverage, and use this understanding as a basis both for accurate description of, and substantive improvement of kernel tests.

2. Environment

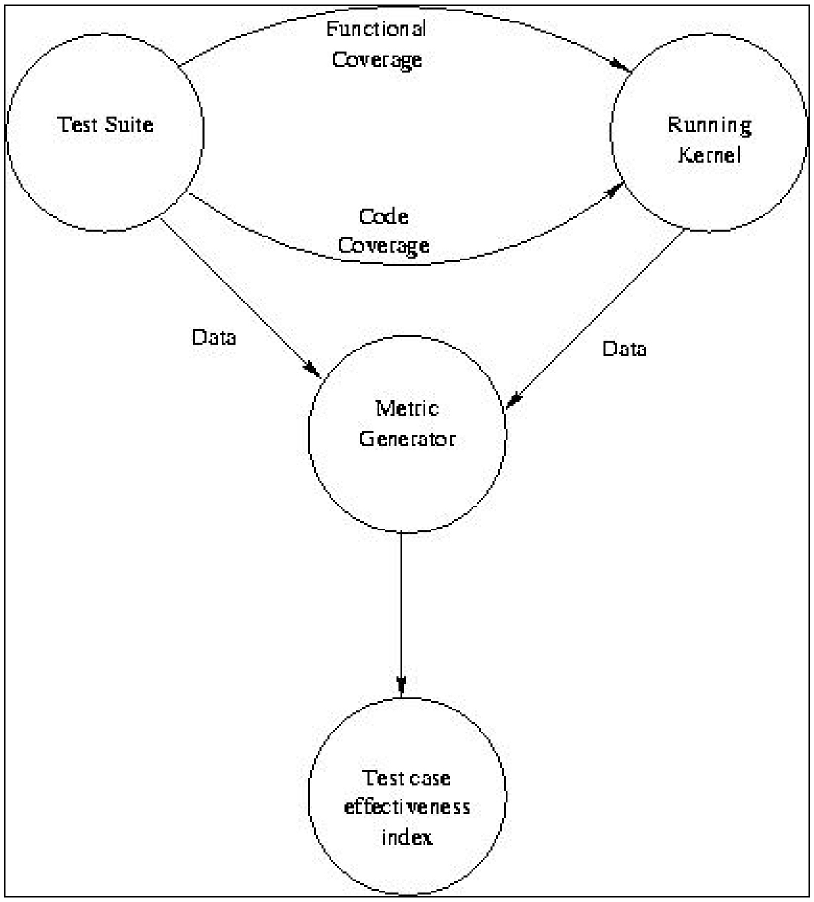
This is a team effort consisting of software engineers at the Linux Technology Center at IBM®. We based this Linux kernel code coverage analysis effort on the Linux 2.4.17 kernel with patches required to enable profiling using gcov, using Debian(TM)(SID) distribution and Intel(TM) Pentium III Coppermine processor. The collection, processing and visual representation of coverage data is automated so that this process can be made repeatable and reproducible by the community on the Linux kernel versions of their choice.

3. Big-Picture Model

We present a unique model to analyze the kernel code coverage using LTP kernel test-suite called the *big picture model.*

By executing the LTP kernel test-suite and analyzing the results of each test execution we were able to provide coverage information with respect to *functional coverage* provided by the test-suite. A program or procedure is functionally correct if it does its intended function in conformance with a well-established set of specifications. Functional coverage is complete if there exists test-cases that test all possible functionality provided by this program or procedure.

An equally important and complementary component of this big picture model is *code coverage.* It is important to be able understand which parts of Linux kernel are being impacted by executing the test-cases.



Big-Picture Model

Code coverage and functional coverage statistics, as well as the size of each component can be used to calculate the effectiveness of the test-cases. A test-case effectiveness index may be generated that represents the measure of kernel coverage provided by the test-cases in the test-suite. This model is illustrated above.

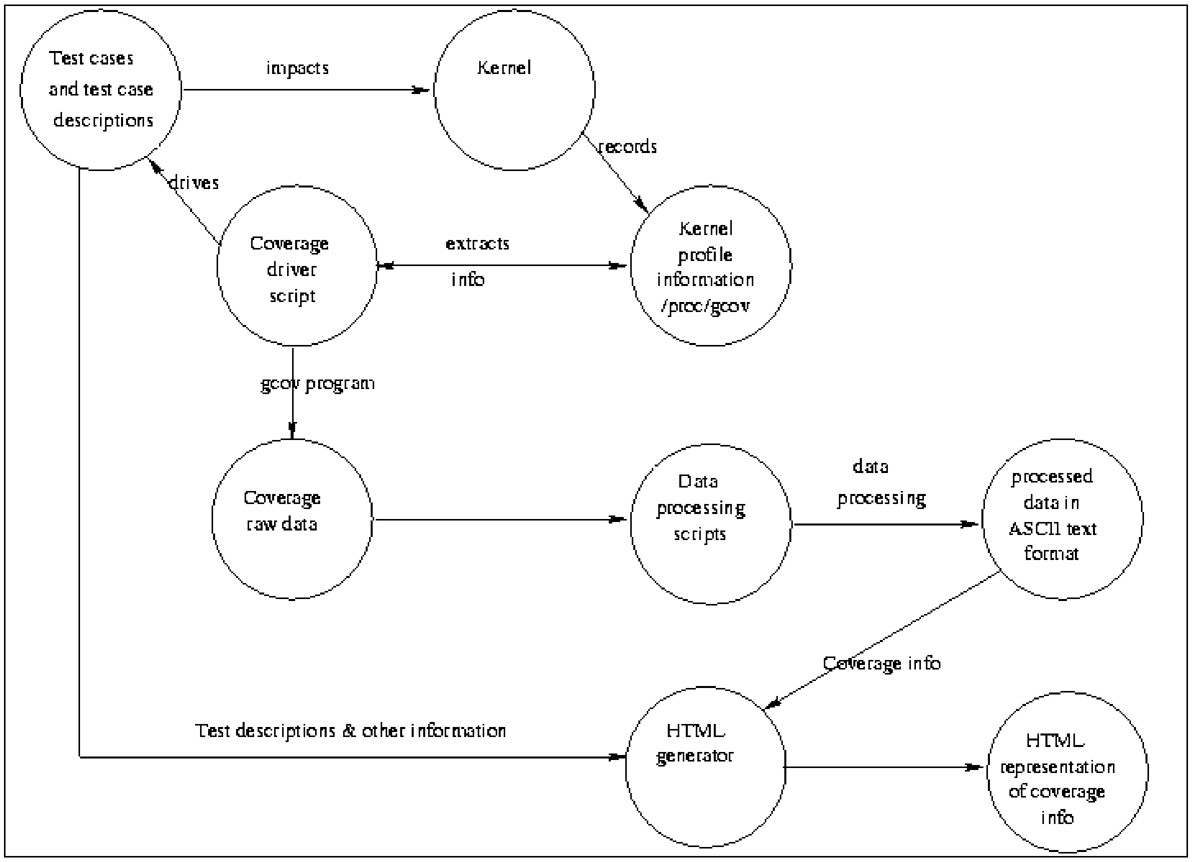
The functional coverage together with the code coverage provides a *big picture* of LTP kernel test-suite's coverage of the Linux kernel. The current effort, Analysis of LTP's kernel code coverage implements only one part of this big-picture model.

The remainder of this document will provide the relevant details on the operational model, overview of tools and data format, and discuss results and their implications.

4. Code Coverage Operational Model

In order to present concise, accurate and meaningful information about the existing coverage we clearly defined. *Coverage data capture* - The activity of obtaining data. *Coverage data processing* - The process of processing captured data. *Coverage data representation* – Presenting the data in an easy to navigate and analyze format. and *Coverage data analysis* - analyzing the implications of the result.

The flow chart below attempts to provide a pictorial view of logical flow of events in the coverage data capture, processing and visualization phases.


Test cases and test case descriptions
impacts
Kernel
records
drives
Coverage driver script
extracts info
Kernel profile information /proc/gcov
gcov program
Coverage raw data
Data processing scripts
data processing
processed data in ASCII text format
Coverage info
Test descriptions & other information
HTML generator
HTML representation of coverage info

Code Coverage Operational Overview

4.1Capture of Code Coverage Data

Collection of coverage data or *coverage data capture* may be done in two ways. 1.Capture code coverage data by running the test-suite as a whole. 2.Identify the impact of individual tests on the kernel or parts of the kernel.

The first approach, although straightforward and easy to implement, does not provide us with fine granularity that might be required by developers. A kernel developer working on the memory management (MM) subsystem of the Linux kernel might require coverage information with respect to tests that impact the memory manager. Taking it a step further, the developer might require coverage information on certain functionality of the memory manager. Understanding the impact of each individual test-case on the kernel is desirable.

Clearly the second approach is the most beneficial. It provides coverage information of each individual test and its impact (or coverage) on the individual files in the kernel. A set of tests can be easily isolated such that they satisfy the developer's requirements. Thus our approach was to collect coverage data for individual kernel subsystems for every test that is available in LTP.

The Linux kernel source tree follows a hierarchy or organization that closely resembles the logical and functional hierarchy of the kernel and its subsystems. The test-cases in the LTP kernel test-suite also follow a certain hierarchy, a *dependency-order*, i.e. strongly connected test-cases are placed together at the bottom of the source tree and the loosely connected tests are placed towards the top of the source tree. We were able to make a one to one relation between individual tests and the kernel subsystem that the test-cases might have an impact on. For example, the test-cases that test various aspects of the file-system are grouped under the *…/ltp/test-cases/kernel/fs* directory in the test-suite. The tests validate, stress or verify the various file-system related functionality. This relationship in organization of the kernel and the test-suite makes it convenient to collect code coverage data per test-case for the entire file-system (fs subsystem) of the kernel.

Collection of data is a three-step procedure. First, the kernel module that produces profiling information, namely *gcov.o,* was loaded. Second, the test-case for which coverage data has to be captured is executed and last, the command *gcov* with appropriate parameters is used to obtain coverage information with respect to that test. These three steps are illustrated below.

1. insmod <*path to module*> /gcov.o

2. ./<*test-case name*>

3. gcov –o /proc/gcov/<*name of the subsystem*> /proc/gcov/<*name of the subsystem*>/<*kernel file to be profiled*>

Step 3 is repeated for each kernel file that belongs to a particular subsystem. The out-put from this step is stored as <*kernel file name*>*.gcov* in the directory from which the data capture scripts were launched.

Coverage data capture technique may be either purely *manual* or purely *automated*. We believe that a purely manual approach is inefficient for high volumes of data that we might have to deal with. It increases the cost in terms of time and labor required making it an impractical approach. A purely automated approach, or the *fire-and-forget* approach, is inherently inflexible, although it makes the task less expensive in terms of time and labor compared to the manual approach. Therefore, we used a *hybrid* approach. We developed tools in C and gawk that take input parameters, providing the flexibility and control over the code coverage data capture process, at the same time this process is fairly cost efficient. The data that is captured is stored as simple ASCII test file, it now can be processed and made available for graphical visualization.

4.2 Processing of Code Coverage Data

The data that was captured is called *raw data*. This raw data, in plain ASCII text format, contains the kernel source code and the frequency of execution of each line of code. For each test we typically had over fifteen to twenty files that had relevant coverage information. This raw data is processed such that we derive concise and meaningful information about coverage, and also convert it some convenient format that is suitable for display and analysis.

The volume of coverage related data generated as a result of executing each test in the LTP kernel test-suite is immensely large. Again a purely manual approach is clearly impractical. We followed a fully automated approach to filtering out relevant data and convert this data to a suitable format. We chose to use the HTML format. HTML format makes it convenient for the target audience to browse through large amounts of information and at the same time is portable. Also, the information can be located on a web server and it can be conveniently accessed by anyone interested in coverage information.

We felt that the collecting data containing the following information is relevant for understanding kernel code coverage:

- Parts of the kernel that are impacted by the test.
- Number of lines of kernel code per kernel file executed as a result of the test.
- Kernel files that are not affected by a particular test in the associated kernel subsystem.
- Frequency of execution of each line of code in the kernel.
- List of distinct lines of code in the kernel file on which test-cases had an impact .

One major issue that we had to consider while determining the test coverage of the kernel subsystem was accuracy of the information. Although it is most desirable to identify exactly which lines of kernel code were impacted by the test, it may not be possible to attain that degree of accuracy. Normal operation of kernel and background jobs that are continuously running will cause gcov to record code in certain kernel files as being impacted. This information must be filtered out. Gcov provides a mechanism by which the counters that keep track of the frequency of execution of each line of source code in the kernel can be reset. Together with this facility, we proposed a simple technique to filter such *noise*. We generated profile information for the entire running kernel up to the point after the profiling module is loaded. We did this simply by talking a snap shot of the */proc/gcov* directory after the profiling module was loaded. This we called *base line raw data*, this base line raw data was then filtered out of the data that was captured after the test was executed.

As an example, for determining kernel coverage for test *mmap01*, the *base line raw data* for the entire kernel was generated, the test was executed and the coverage data was captured. The area of interest here is the MM subsystem. As a result of executing mmap01, if line number N in the kernel file mmap.c was executed X times, and the same line was executed Y times as per information in the *base line raw data*, we evaluated the number of time the line was executed as a result of the test as (Y – X), i.e. by subtracting the baseline from the actual data. This, in our opinion, is a fairly accurate statistic.

4.3 Analysis of Code Coverage Data

Analysis of the data gathered an open-ended issue, its interpretation is purely dependent on the perspective of the audience of the data we present in HTML. We envision the requirement for two separate approaches to analyze the data obtained on kernel coverage namely, *Deductive analysis* and *retrospective analysis.*

Coverage was measured in terms of the number of distinct lines of C code in the kernel that were executed as a result of running the tests in the test-suite. We used the following scale and color scheme to represent kernel code coverage by LTP’s kernel test-suite .

10 % or less covered
10% to 40%
40% or more

No coverage is indicated by color red, 10% to 40% covered by yellow, 40 % or more by green.

It will be useful to gather similar data using different kernel versions, distributions and hardware platforms, and perform a retrospective analysis of coverage on this empirically gathered information. This is considered as future work.

5. Overview Of Tools And Data Format

5.1 Data Capture Tools

Kernel Module to Collect Code Coverage Data

Gcov is an open source coverage analysis program. Gcov support can be enabled in the kernel by patching the kernel with appropriate gcov release and compiling it as a module. It produces basic block profile information that contains the following statistics per source file for the running kernel and modules:

how often each line of code executes
what lines of code are actually executed
how much computing time each section of code uses

Template Generator

This bash shell script takes a snapshot of all the files under /proc/gcov directory. These files constitute our baseline raw data. Flow Char 1.0 illustrates the program logic.

Inputs:	<file_name>.gcov (gcov output) (Table 1.1)
Output:	<file_name>.gcov.tmpl (baseline raw data)
Other Programs called:	gcov

Program to Process Generated Data and Report Statistics

This is a C program that counts the frequency of execution of each line. It produces the following data. Flow Chart 1.2 illustrates the program logic.

Inputs:	1. <file_name>.gcov.tmpl (baseline raw data) 2. <file_name>.gcov (coverage output)
Output:	1. Frequency of execution of each line of code 2. Line number information
Other Programs called:	NONE

Test Driver

This is the driver bash-shell script that is responsible for running each test and generating kernel profile information. Flow Chart 1.3 illustrates the program logic.

Input:	1. Test-case 2. Coverage data generated by gcov 3. baseline raw data.
Output	Processed data in meta language. <filename>.kernel_subsystem.info (Table 1.2)
Other Programs called.	1. gcov 2. Program to Process Generated Data and Report Statistics

Test Description generator

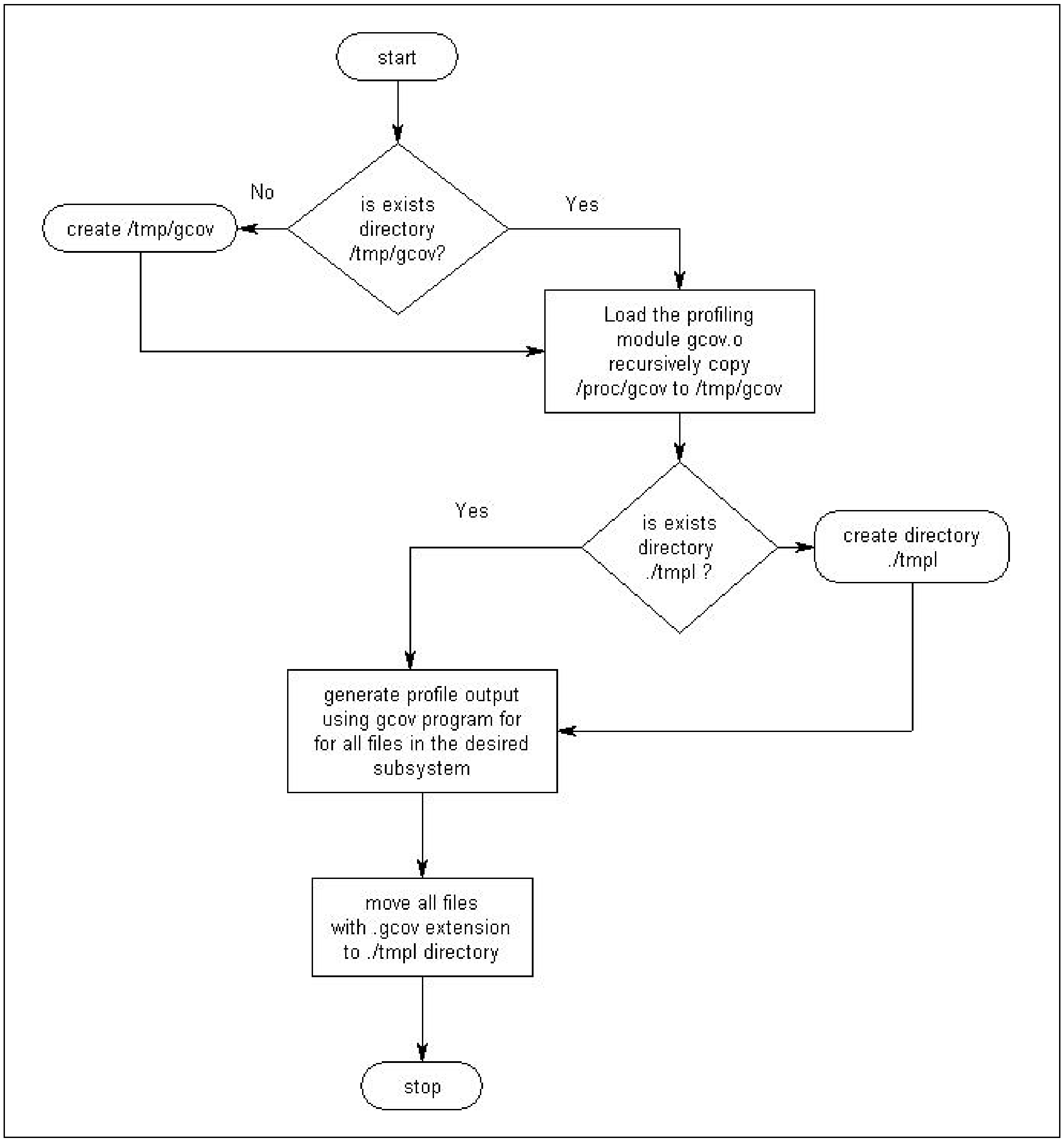
AWK script that generates the description of each test-case in meta language. Flow Chart 1.4 illustrates the program logic.

Inputs:	ASCII file containing the test name and its description.(Table 1.3)
Output:	ASCII file containing test name and description in meta language.(Table 1.4)
Other Programs called:	NONE

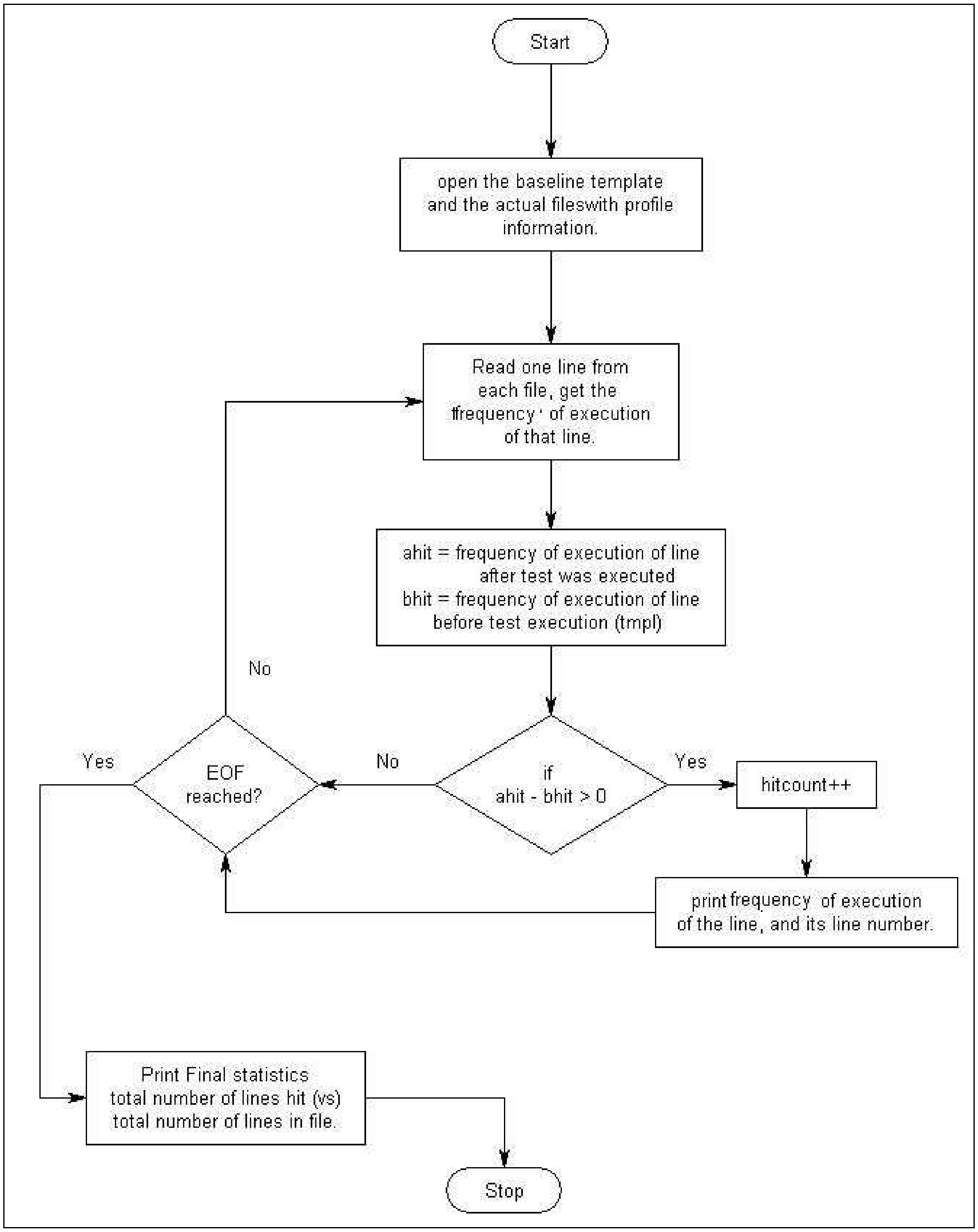
HTML Generator

PERL script that generates HTML files from the coverage data in meta language format.

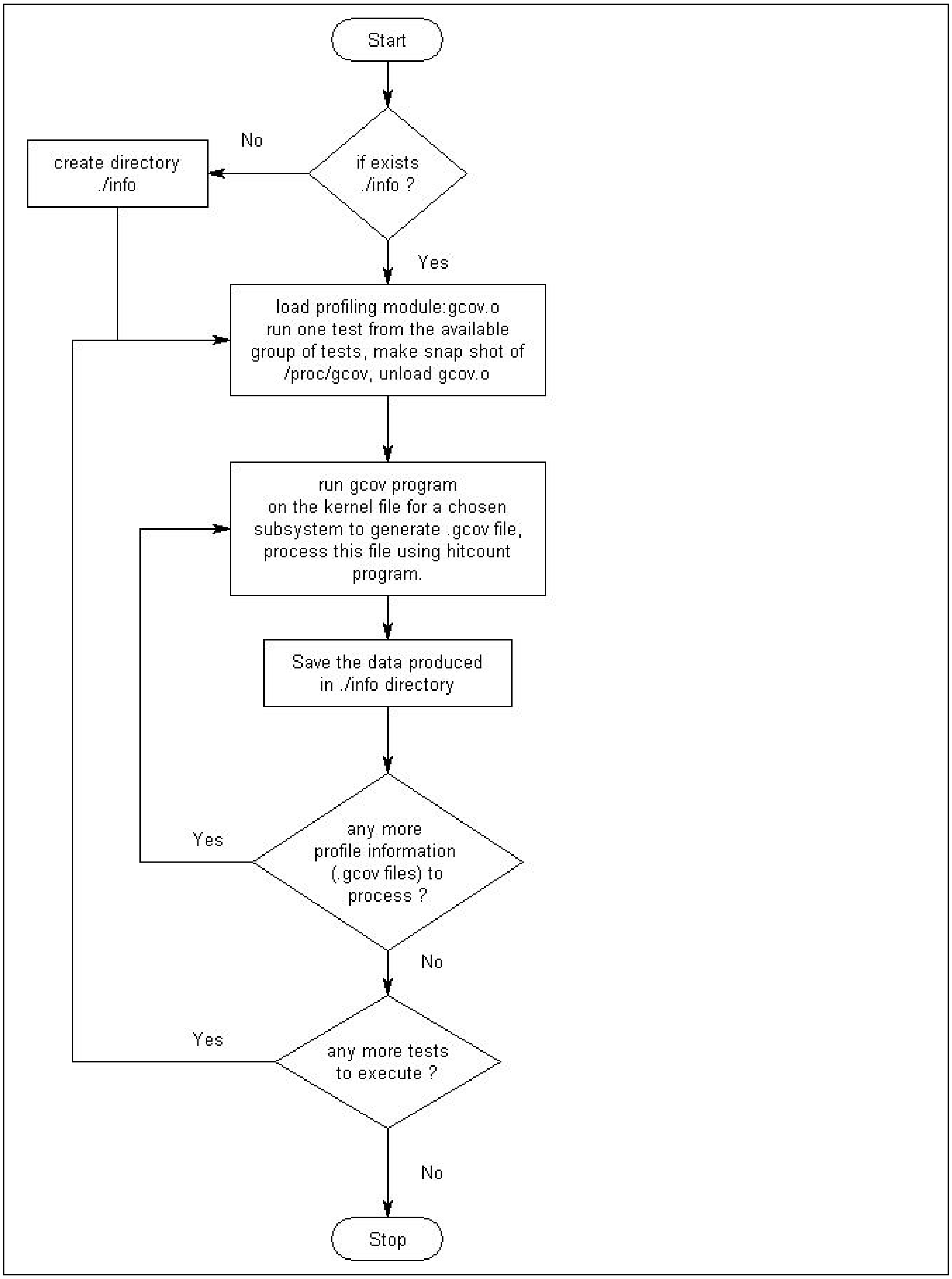
Inputs:	1. Test-case description files 2. Coverage information in meta language format. <filename>.<kernel_subsystem>.info(Table 1.4)
Output:	HTML files.
Other Programs called:	NONE


start
is exists directory /tmp/gcov?
No
Yes
create /tmp/gcov
Load the profiling module gcov.o recursively copy /proc/gcov to /tmp/gcov
is exists directory ./tmpl ?
Yes
create directory ./tmpl
generate profile output using gcov program for for all files in the desired subsystem
move all files with .gcov extension to ./tmpl directory
stop

Flow Chart 1.0: Describes logic used to generate template files.

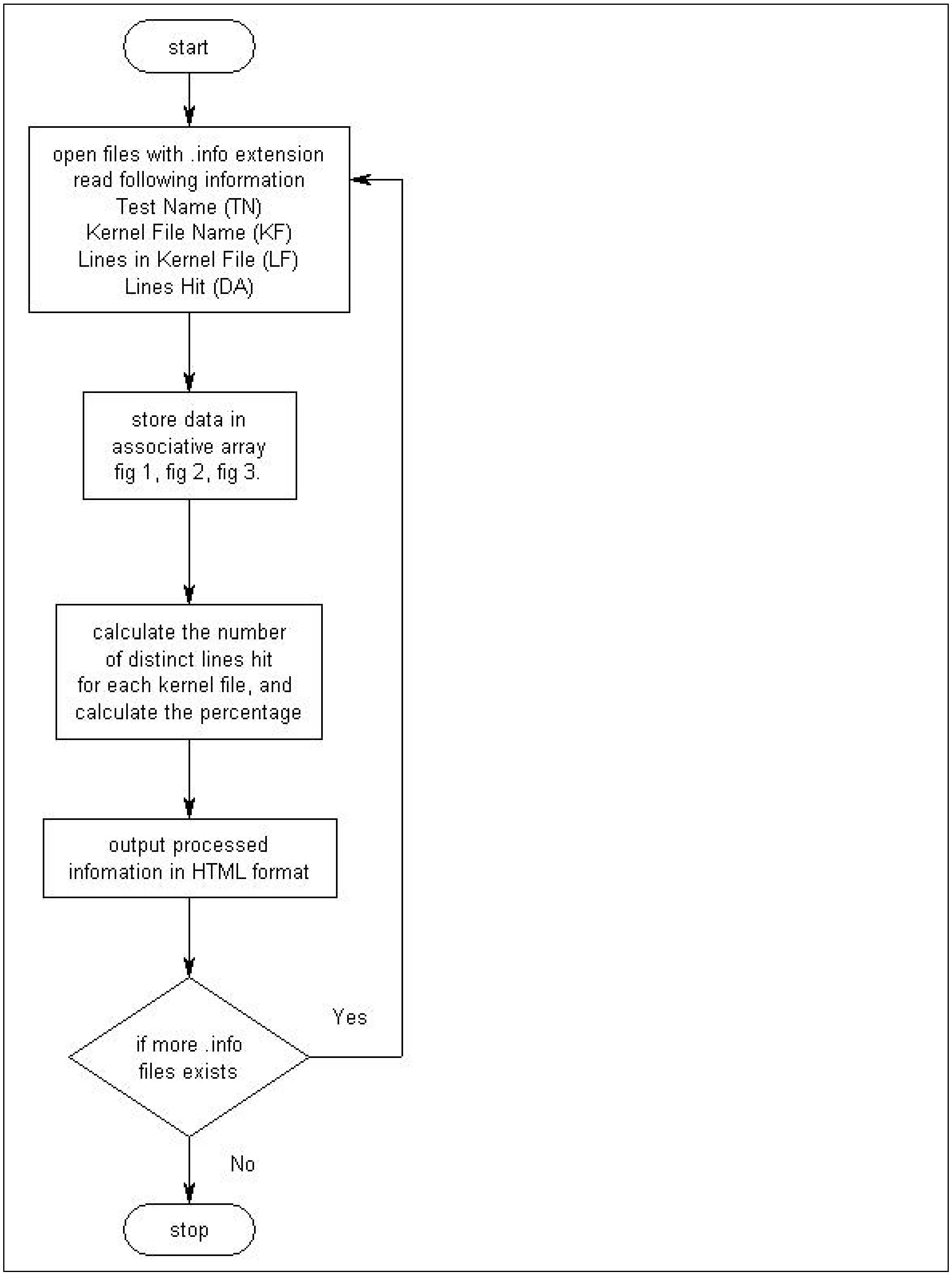

Start
open the baseline template and the actual fileswith profile information.
Read one line from each file, get the 'frequency ' of execution of that line.
ahit = frequency of execution of line after test was executed
bhit = frequency of execution of line before test execution (tmpl)
if ahit - bhit > 0
Yes
hitcount++
print frequency of execution of the line, and its line number.
No
EOF reached?
No
Yes
Print Final statistics total number of lines hit (vs) total number of lines in file.
Stop

Flow Chart 1.1 Describes program logic used in hitcount program.



Start
if exists ./info ?
No
create directory ./info
Yes
load profiling module:gcov.o run one test from the available group of tests, make snap shot of /proc/gcov, unload gcov.o
run gcov program on the kernel file for a chosen subsystem to generate .gcov file, process this file using hitcount program.
Save the data produced in ./info directory
any more profile information (.gcov files) to process ?
Yes
No
any more tests to execute ?
Yes
No
Stop

Flow Chart 1.3 Describes the program logic of the driver



Flow Chart 1.4 Describes the program logic for generating HTML.

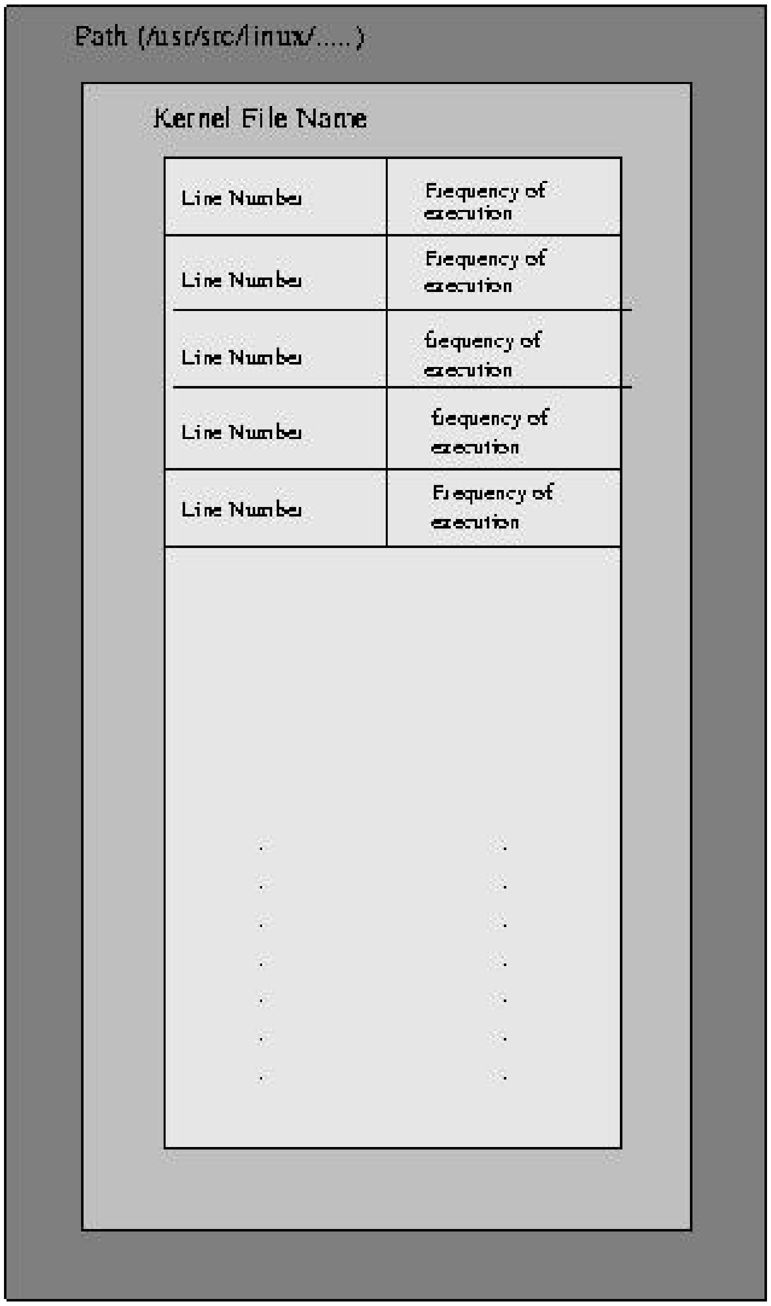


Figure 1.0 Associative Array of Line numbers and Frequency of execution of that line

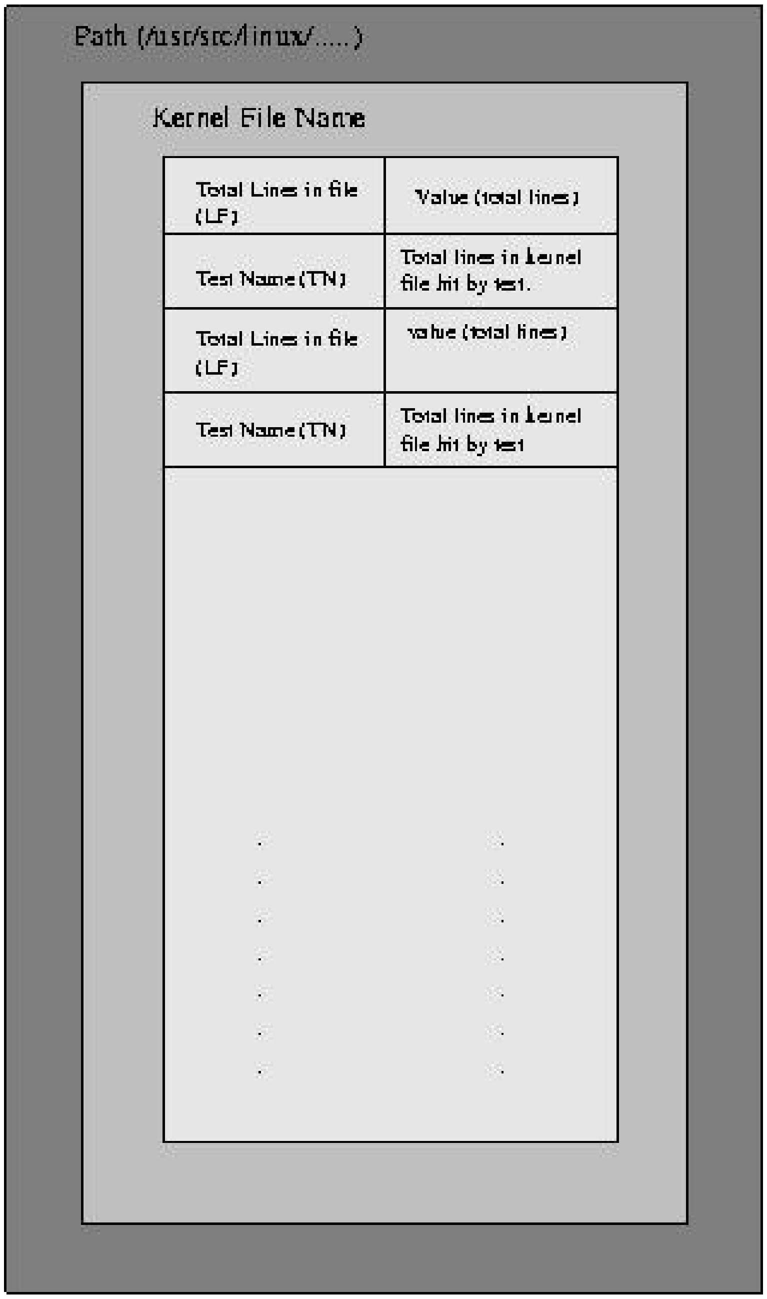
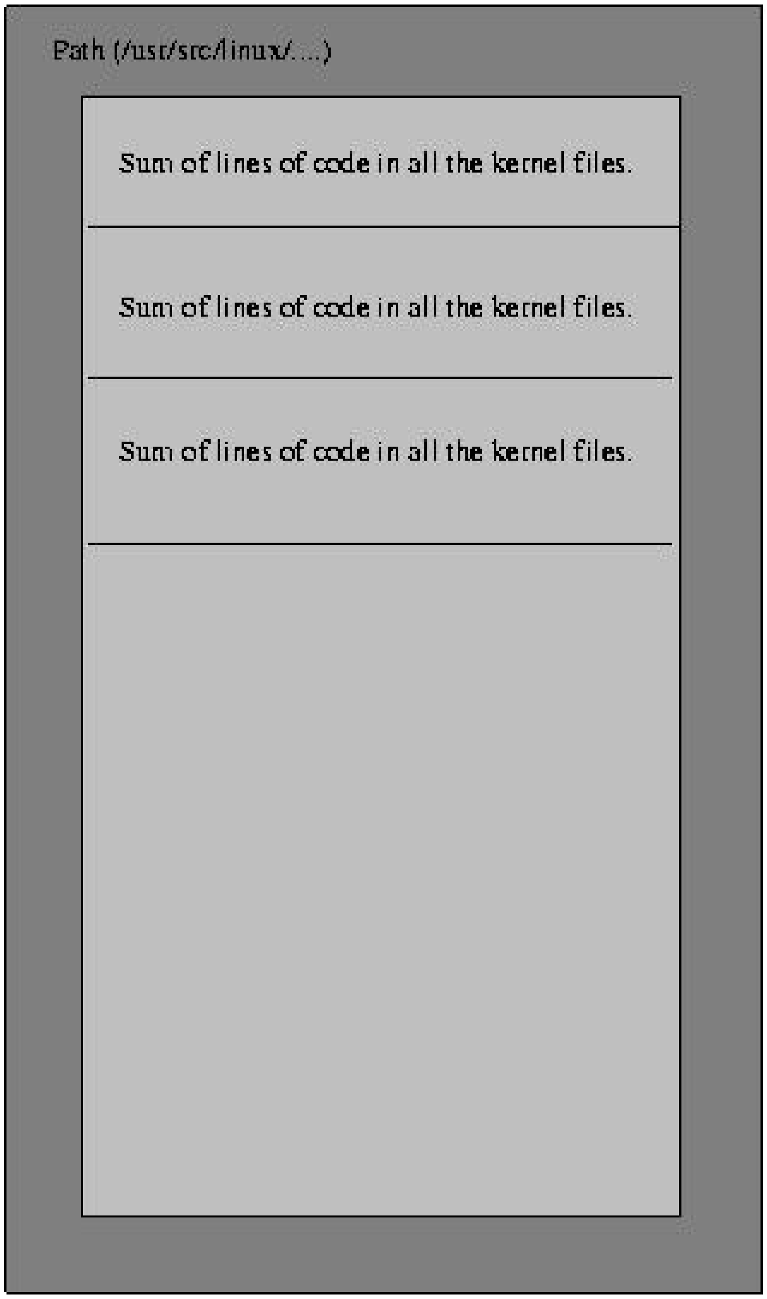


Figure 1.1 Associative Array of Test names and Number of lines hit in the kernel file.


Path (/usr/src/linux/....)
Sum of lines of code in all the kernel files.
Sum of lines of code in all the kernel files.
Sum of lines of code in all the kernel files.
Figure 1.1 Scalar Array of Sum of lines of code in all the kernel files in that path.

5.2 Data Format

```
             static inline void copy_cow_page(struct page * from, struct page
   2125368   {
   2125368         if (from == ZERO_PAGE(address)) {
    141696               clear_user_highpage(to, address);
    141696               return;
   1983672         }
   1983672         copy_user_highpage(to, from, address);
   2125368   }

             mem_map_t * mem_map;

            /*
             * Called by TLB shootdown
             */
             void __free_pte(pte_t pte)
             {
  57187850          struct page *page = pte_page(pte);
  57187850          if ((!VALID_PAGE(page)) || PageReserved(page))
     39167               return;
  57148683          if (pte_dirty(pte))
  13774904                 set_page_dirty(page);
  57148683          free_page_and_swap_cache(page);
  57148683    }
```

Table 1.0 Code fragment from memory.c.gcov. Out put from profiling tool: gcov.

Executing the gcov command on the desired kernel file in the */proc/gcov* directory produces a file with a *.gcov* extension. For example Table 1.1 shows a code fragment from the file memory.c.gcov that was produced as a result of executing one of the memory test and capturing coverage data. The command used for generating this file was gcov –o /proc/gcov/mm /proc/gcov/mm/memory.c.

The numbers on the left hand side of the table indicate the number of times that particular line of code was executed. For instance the following line:

```
 57187850           if ((!VALID_PAGE(page)) || PageReserved(page))
```

means the code fragment was executed 57187850 number of times.

```
TN: mmap01
KF: /usr/src/linux/mm/memory.c
DA: 63,1178977
DA: 64,1178977
DA: 65,81019
DA: 66,81019
DA: 67,1097958
.      .      .
.      .      .
.      .      .
DA: 1427,12353853
DA: 1428,12353853
LH: 430
LF: 1445
end_of_record
KF: /usr/src/linux/mm/mmap.c
DA: 64,1550832
DA: 67,1550832
DA: 71,1550832
DA: 72,1550832
DA: 73,1550832
.      .      .
.      .      .
.      .      .
DA: 1170,65872
DA: 1171,65872
DA: 1172,65872
LH: 541
LF: 1172
end_of_record
KF: /usr/src/linux/mm/filemap.c
DA: 74,492921
DA: 76,492921
DA: 77,492921
DA: 78,492921
.      .      .
.      .      .
```

Table 1.2 Data file fragment: mmap01.mm.info. Output of test driver; bash-shell script.

The test driver, a bash-shell script, generated this Data file fragment. The script takes the raw data file (table 1.1) and produces the output shown in table 1.2. This file is understood by the HTML generator (PERL script).

In the above table (Table 1.2) the following conventions are used:

TN: Name of the test.

KF: Name of the kernel file that this test had an impact on.
DA: Data element, DA: <num1>,<num2> where num1 is the line number in the kernel source file, and num2 is the frequency with which it was executed.
LH: Number of lines in the kernel file that were executed.
LF: Number of lines in the file.
end_of_record
The name of the output file follows the following format:
Name_of_the_kernel_file.kernel_sub_system.info

```
mmstress

    Performs General Stress with Race conditions

mmap1

    Test the LINUX memory manager. The program is aimed at
    stressing the memory manager by simultanious map/unmap/read
    by light weight processes, the test is scheduled to run for
    a mininum of 24 hours.

mmap2

    Test the LINUX memory manager. The program is aimed at
    stressing the memory manager by repeaded map/write/unmap of a
    of a large gb size file.

mmap3

    Test the LINUX memory manager. The program is aimed at
    stressing the memory manager by repeaded map/write/unmap
    of file/memory of random size (maximum 1GB) this is done by
    multiple processes.
```

Table 1.3 Test-case description file.

Table 1.3 is a fragment of the test-case description file. It contains the name of the test and a brief description of the test-case. Each test in the LTP test-suite is documented in a test description file, maintained by the maintainer of LTP.

```
TN: mmstress
TD: Performs General Stress with Race conditions
TN: mmap1
TD: Test the LINUX memory manager. The program is aimed at
TD: stressing the memory manager by simultanious map/unmap/read
TD: by light weight processes, the test is scheduled to run for
TD: a mininum of 24 hours.
TN: mmap2
TD: Test the LINUX memory manager. The program is aimed at
TD: stressing the memory manager by repeaded map/write/unmap of a
TD: of a large gb size file.
TN: mmap3
TD: Test the LINUX memory manager. The program is aimed at
TD: stressing the memory manager by repeaded map/write/unmap
TD: of file/memory of random size (maximum 1GB) this is done by
TD: multiple processes.
TN: mmap001
```

Table 1.4 Data file: formatted by gawk script

The description file is formatted such that the name of the test-case and its description is tagged. The HTML generator uses this file.

In the Table 1.4 the following conventions are used:

TN: Name of the test-case

TD: Brief description of the test-case.

6. Results and their implications

Linux Kernel Subsystem (As given in source tree)	**Overall Coverage**
/usr/src/linux/arch/i386/kernel	9.00%
/usr/src/linux/arch/i386/mm	11.65%
/usr/src/linux/fs	29.51%
/usr/src/linux/fs/autofs4	0.00%
/usr/src/linux/fs/devpts	3.31%
/usr/src/linux/fs/ext2	28.24%
/usr/src/linux/fs/isofs	0.00%
/usr/src/linux/fs/lockd	0.13%
/usr/src/linux/fs/nfs	0.00%
/usr/src/linux/fs/nfsd	0.00%
/usr/src/linux/fs/partitions	0.00%
/usr/src/linux/fs/proc	25.44%
/usr/src/linux/ipc	46.02%
/usr/src/linux/kernel	25.63%
/usr/src/linux/mm	24.50%
/usr/src/linux/net	48.40%
/usr/src/linux/net/802	0.00%
/usr/src/linux/net/core	12.87%
/usr/src/linux/net/ethernet	28.21%
/usr/src/linux/net/ipv4	18.37%
/usr/src/linux/net/netlink	0.00%
/usr/src/linux/net/packet	0.00%
/usr/src/linux/net/sched	3.36%
/usr/src/linux/net/sunrpc	0.00%
/usr/src/linux/net/unix	47.80%

Table 1.4 LTP kernel test-suite's Linux kernel code coverage

The Table 1.5 summarizes our findings; clearly certain parts of the kernel are not impacted by any of the tests. This may be due to three reasons, 1. Tests not available, 2. Test is not exercising most of the functionality provided by this part of the kernel. 3. Or certain tests were not executed; this code coverage analysis used all the tests listed in the runalltests.sh script, the networking tests were not executed. For instance, The LTP's kernel test-suite appears to have had no impact on the /usr/src/linux/net/sunrpc subsystem of the kernel. By careful inspection of the test-suite we can associate this 0.00% code coverage to the lack of tests. It is also important to take into account the size of each subsystem, for instance the file-system component is larger in size compared to the memory manager subsystem of the kernel, it might not be realistic to require the tests that provide adequate coverage for the entire file-system component of the kernel. We recommend that the audience use their judgment while drawing conclusion with respect to code coverage. This is where the big picture model will play an important role, this model will take into account the size, code coverage, functionality coverage etc while generating the test-case effectiveness index, this is future work.

The kernel subsystems colored in *red* is our area of focus. Although creating automated tests that integrate well with the LTP kernel test-suite may not be possible in certain cases, it is certainly the starting point for anyone who wishes to work on improving the coverage provided by LTP's kernel test-suite.

The test-cases that test the kernel subsystems colored in *yellow* might be improved further such that they provide adequate coverage. The HTML document provides a list of tests and a brief description of the tests that impact these parts of the kernel. We will encourage the members of the community to contribute new tests or enhance the existing tests, such that the test-suite is able to provide adequate coverage for these kernel subsystems.

We claim that the test-cases for the kernel subsystems colored in green as *good enough* as they provide adequate coverage for the particular kernel subsystem. It is desirable to investigate further in order that the code coverage provided by these tests can be further improved, although, the main focus must be on improving those tests that provide less than adequate coverage.

7. Summary and Future Work

With the existing tools and the ones we have developed, we have established an effective framework for capturing and analyzing kernel code coverage information, and also proposed a model (big-picture model) to calculate the kernel coverage provided by these tests. We have provided an interactive and easy to use and reproduce kernel coverage data capture and analysis method. Our work is thus a foundation for future work in similar areas.

We have provided a graphical front end that will enable users to explore the relationship between the tests and various kernel subsystems, and their interactions. While we will not claim that our visualization technique is unique, we do claim that we found a technique that provides insight and better understanding of kernel coverage using the LTP test-suite.

In our attempt to provide a simple mechanism to comprehend coverage of the Linux kernel using LTP kernel test-suite, we clearly identified areas that need improvement and areas where the tests provide ample coverage. In addition to the immediate utility of this analysis effort, it opens up avenues for contribution from the community for a varity of improvements and additions to the LTP kernel test-suite and the Linux kernel itself.

We have identified certain key areas where further contributions might be of significant importance.

> Implement the big-picture model by calculating the test-case effectiveness index for each test-case and measure the LTP kernel test-suites coverage of the Linux kernel. This in our opinion will be a complete and true indicator of the coverage provided by the test-suite.
>
> LTP kernel test-suite and the Linux kernel itself have multiple platform support. Processor architecture supported include Intel 32bit, Intel 64bit, Power PC 32bit, PowerPC 64bit, S390, etc. Our focus has been on Intel 32bit platforms. This coverage analysis in particular was done on an Intel 32bit machine. Similar work on other platforms that run ports of the Linux kernel might be of great relevance.
>
> We have observed certain interesting behaviors when the tests were executed on a SMP machine versus UP machine. The kernel code that gets built into the kernel is also different for an SMP and UP kernel. SMP kernel coverage analysis is another area of paramount significance.
>
> We have chosen to use the 2.4.17 version of the Linux kernel. It might be useful to reproduce similar analysis work on other stable releases and also on unstable or test kernels, this will provide the developers who use LTP kernel test-suite as a verification tool to make better choice of tests to perform sanity checks on code changes in the subsystem under development.

Perform code coverage analysis for other test-suites that are available in the LTP. Networking tests, database tests (DOTS) etc.

Automated execution of test cases that will impact certain subsystem of the kernel.

Acknowledgements and Contibutions

This work would not have been a success without the unflagging support from Linda Scott and Ginny Black, providing us with technical and managerial, leadership and coordination. Airong Zhang was instrumental in early stages, collecting and formatting data on test-cases available in the LTP kernel test-suite.

A special acknowledgement for the contributions from Megan Bock and James Kenefick who, designed and implemented the PERL script that generated HTML pages from kernel coverage data collected in ASCII format.

I also thank every team member and colleague who reviewed this document and provided us with useful comments and suggestions.

DISCLAIMER

Document Author:
Manoj Iyer